Model Checking with Multi-Threaded IC3 Portfolios

Sagar Chaki and Derrick Karimi

Software Engineering Institute, Carnegie Mellon University
{chaki,dhkarimi}@sei.cmu.edu

Abstract. Three variants of multi-threaded IC3 are presented. Each variant has a fixed number of IC3s running in parallel, and communicating by sharing lemmas. They differ in the degree of synchronization between threads, and the aggressiveness with which proofs are checked. The correctness of all three variants is shown. The variants have unpredictable runtime. On the same input, the time to find the solution over different runs varies randomly depending on the thread interleaving. The use of a portfolio of solvers to maximize the likelihood of a quick solution is investigated. Using the Extreme Value theorem, the runtime of each variant, as well as their portfolios is analysed statistically. A formula for the portfolio size needed to to achieve a verification time with high probability is derived, and validated empirically. Using a portfolio of 20 parallel IC3s, speedups over 300 are observed compared to the sequential IC3 when on hardware model checking competition examples.

1 Introduction

In recent years, IC3 [5] has emerged as a leading algorithm for model checking hardware. It has been refined [8] and incorporated into state-of-the-art tools, and generalized to verify software [10, 6]. Our interest is that IC3 is amenable to parallelization [5], and promises new approaches to enhance the capability of model checking by harnessing the abundant computing power available today. Indeed, the original IC3 paper [5] described a parallel version of IC3 informally and reported on its positive performance. In this paper, we build on that work and make three contributions.

First, we formally present three variants – IC3SYNC, IC3ASYNC and IC3PROOF – of parallel IC3, and prove their correctness. All the variants have some common features: (i) they consist of a fixed number of threads that execute in parallel; (ii) each thread learns new lemmas and looks for counterexamples (CEXes) or proofs as in the original IC3; (iii) all lemmas learned by a thread are shared with the other threads to limit duplicate work; and (iv) if any thread finds a CEX, the overall algorithm declares the problem unsafe and terminates.

However, the variants differ in the degree of inter-thread synchronization, and the frequency and technique for detecting proofs, making different trade-offs between the overhead and likelihood of proof-detection. Threads in IC3SYNC (cf. Sec. 3.1) synchronize after each round of new lemma generation and propagation,

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **15 JAN 2015**	2. REPORT TYPE **N/A**	3. DATES COVERED
4. TITLE AND SUBTITLE **Model Checking with Multi-Threaded IC3 Portfolios**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Karimi /Sagar Chaki Derrick**		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S) 11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited.

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**SAR**	**21**	

and check for proofs in a centralized manner. Threads in IC3ASYNC (cf. Sec. 3.2) are completely asynchronous. Proof-detection is decentralized and done by each thread periodically. Finally, threads in IC3PROOF are also asynchronous and perform their own proof detection, but more aggressively than IC3ASYNC. Each thread saves the most recent set of inductive lemmas constructed. When a thread finds a new set of inductive lemmas, it checks if the collection of inductive lemmas across all threads form an inductive invariant. In order of increasing overhead (and likelihood) of proof-detection, the variants are: IC3SYNC, IC3ASYNC, and IC3PROOF. Collectively, we refer to the variants as IC3PAR.

The runtime of IC3PAR is unpredictable (this is a known phenomenon [5]). In essence, the number of steps to arrive at a proof (or CEX) varies with the thread interleaving. We propose to counteract this variance using a portfolio – run several IC3PARs in parallel, and stop as soon as any one terminates with an answer. But how large should such a portfolio be? Our second contribution is a statistical analysis to answer this question. Our insight is that the runtime of IC3PAR should follow the Weibull distribution [18] closely. This is because it can be thought of as the *minimum* of the runtimes of the threads in IC3PAR, which are themselves independent and identically distributed (i.i.d.) random variables. According to the Extreme Value theorem [9], the minimum of i.i.d. variables converges to a Weibull. We empirically demonstrate the validity of this claim.

Next, we hoist the same idea to a portfolio of IC3PARs. Again, the runtime of the portfolio should be approximated well by a Weibull, since it is the minimum of the runtime of each IC3PAR in the portfolio. Under this assumption, we derive a formula (cf. Theorem 5) to compute the portfolio size sufficient to solve any problem with a specific probability and speedup compared to a single IC3PAR. For example, this formula implies that a portfolio of 20 IC3PARs has 0.99999 probability of solving a problem in time no more than the "expected time" for a single IC3PAR to solve it. We empirically show (cf. Sec. 5.2) that the predictions based on this formula have high accuracy. Note that each solver in the portfolio potentially searches for a different proof/CEX. The first one to succeed provides the solution. In this way, a portfolio utilizes the power of IC3 to search for a wide range of proofs/CEXes without sacrificing performance.

Finally, we implement all three IC3PAR variants, and evaluate them on benchmarks from the 2014 Hardware Model Checking Competition (HMCC14) and "TIP". Using each variant individually, and in portfolios of size 20, we observe that IC3PROOF and IC3ASYNC outperform IC3SYNC. Moreover, compared to a purely sequential IC3, the variants are faster, providing an average speedup of over 6 and a maximum speedup of over 300. We also show that widening the proof search of IC3 by randomizing its SAT solver is not as effective as parallelization. Complete details are presented in Section 5.1.

Related Work. The original IC3 paper [5] presents a parallel version informally, and shows empirically that parallelism can improve verification time. Our IC3PAR solvers were inspired by this work, but are different. For example, the parallel IC3 in [5] implements clause propagation by first distributing learned clauses over all solvers and then propagating them one frame at a time, in lock step. It also introduces uncertainty in the proof search by randomizing the back-end SAT solver. Our IC3PAR solvers perform clause propagation asynchronously, and use deterministic SAT solvers. We also present each IC3PAR variant formally with pseudo-code and prove their correctness. Finally, we perform a statistical analysis of the runtimes of both IC3PAR solvers and their portfolios. Experimental results (cf. Sec. 5.1) indicate that a portfolio of IC3PAR solvers is more efficient than a portfolio composed of IC3 solvers with randomized SAT solvers.

A number of projects focus on parallelizing model checking [11, 4, 15, 2, 3, 1]. Ditter et al. [7] have developed GPGPU algorithms for explicit-state model checking. They do not report on variance in runtime, nor analyse it statistically like us, or explore the use of portfolios. Lopes et al. [13] do address variance in runtime of a parallel software model checker. However, their approach is to make the model checker's runtime more predictable by ensuring that the counterexample generation procedure is deterministic. They also do not perform any statistical analysis or explore portfolios.

Portfolios have been use successfully in SAT solving [20, 17, 12, 14], SMT solving [19] and symbolic execution [16]. However, these portfolios are composed of a heterogeneous set of solvers. Our focus is on homogeneous portfolios of IC3PAR solvers and statistical analysis of their runtimes.

2 Preliminaries

Assume Boolean state variables V, and their primed versions V'. A verification problem is (I, T, S) where $I(V)$, $T(V, V')$ and $S(V)$ denote initial states, transition relation and safe states, respectively. We omit V when it is clear from the context, and write S' to mean $S(V')$. Let $Post(X)$ denote the image of X under the transition relation T. Let $Post^k(X)$ be the result of applying $Post(\cdot)$ k times on X with $Post^0(X) = X$, and $Post^{k+}(X) = \bigcup\limits_{j \geq k} Post^j(X)$. The problem (I, T, S) is safe if $Post^{0+}(I) \subseteq S$, and unsafe (a.k.a. buggy) otherwise.

A random variable X has a Weibull distribution with shape k and scale λ, denoted $X \sim \textsc{wei}(k, \lambda)$, iff its probability density function (pdf) f_X and cumulative distribution function (cdf) F_X are defined as follows:

$$f_X(x) = \begin{cases} \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1} e^{-(\frac{x}{\lambda})^k} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \qquad\qquad F_X(x) = 1 - e^{-(\frac{x}{\lambda})^k}$$

Let $X_1, \ldots, X_n$ be i.i.d. random variables (rvs) whose pdfs are lower bounded at zero, i.e., $\forall x < 0 \centerdot f_{X_i}(x) = 0$. Then, by the Extreme Value theorem [9] (EVT), the pdf of the rv $X = \min(X_1, \ldots, X_n)$ converges to a Weibull as $n \to \infty$.

3 Parallelizing IC3

We begin with a description of the sequential IC3 algorithm. Fig. 1 shows its pseudo-code. IC3 works as follows: (i) checks that no state in $\neg S$ is reachable

```
1 //-- global variables
2 var (I,T,S) : problem (P)
3 var F: frame[] (array of frames)
4 var K: int (size of F)
5 var bug: bool (CEX flag)
6
7 //-- invariants
8 ∀i ∈ [0,K−1], let f(i) = ⋀_{j∈[i,K−1]} ⋀_{α∈F[j]} α
9 A_1 : ∀i ∈ [0,K−1] . I ⟹ f(i)
10 A_2 : ∀i ∈ [0,K−2] . f(i) ∧ T ⟹ f'(i+1)
11 A_3 : ∀i ∈ [0,K−3] . f(i) ∧ T ⟹ S'
12 A_4 : ∀i ∈ [0,K−2] . f(i) ∧ T ⟹ S'
13
14 //-- main function.
15 bool IC3 ()
16   if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S' ≠ ⊥)
17     return ⊥;
18   K := 3; F[0] := I; F[1] := ∅;
19   F[2] := ∅; bug := ⊥;
20   while (⊤)
21     @INV{𝓘_1 : A_1 ∧ A_2 ∧ A_3}
22     strengthen(F,K);
23     @INV{𝓘_2 : bug ∨ (A_1 ∧ A_2 ∧ A_4)}
24     if (bug) return ⊥;
25     @INV{𝓘_3 : A_1 ∧ A_2 ∧ A_4}
26     propagate(F,K);
27     if (∃i ∈ [1,K−2] . F[i] = ∅)
28       return ⊤;
29     @INV{𝓘_3}
30     F[K] := ∅; K := K+1;
31 //-- add new lemmas to frames. stop
32 //-- with a CEX or when A_4 holds.
33 void strengthen (F,K)
34   var PQ : priority queue
35   while (⊤)
36     if (f(K−2) ∧ T ⟹ S') return;
37     let m ⊨ f(K−2) ∧ T ∧ ¬S';
38     PQ.insert(m, K−3);
39     while (¬PQ.empty())
40       (m,l) := PQ.top();
41       if (f(l) ∧ T ∧ m' = ⊥)
42         F[l+1] := F[l+1] ∪ {¬m};
43         PQ.erase(m,l);
44       else if (l = 0)
45         bug := ⊤; return;
46       else
47         let m' ⊨ f(l) ∧ T ∧ m;
48         PQ.insert(m', l−1);
49
50 //-- push inductive clauses forward.
51 //-- check for proof of safety.
52 void propagate(F,K)
53   for i : 1...K−2
54     for α ∈ F[i]
55       if (f(i) ∧ T ⟹ α')
56         F[i+1] := F[i+1] ∪ {α};
57         F[i] := F[i] \ {α};
```

Fig. 1. Pseduo-Code for IC3. Variables are passed by reference.

in 0 or 1 steps from some state in I (lines 16–17); (ii) iteratively construct an array of frames, each consisting of a set of clauses, as follows: (a) initialize the frame array and flags (lines 18–19); (b) `strengthen()` the frames by adding new clauses (line 22); if a counterexample is found in this step (indicated by *bug* being set), IC3 terminates (line 24); (c) otherwise, `propagate()` clauses that are inductive to the next frame (line 26); if a proof of safety is found (indicated by an empty frame), IC3 again terminates (lines 27–28); (d) add a new empty frame to the end of the array (line 30) and repeat from step (b).

Definition 1 (Frame Monotonicity). *A function is frame monotonic if at each point during its execution,* $\forall i \in [0, K-1] \,.\, f(i) \implies \tilde{f}(i)$ *where* $\tilde{f}(i)$ *is the value of* $f(i)$ *when the function was called.*

Correctness. Fig. 1 also shows the invariants (indicated by `@INV`) before and after `strengthen()` and `propagate()`. Since `strengthen()` always adds new lemmas to frames, it is frame monotonic, and hence it maintains A_1 and A_3. It also maintains A_2 since a new lemma $\neg m$ is added to frame $F[l+1]$ (line 42) only if $f(l) \wedge T \implies \neg m'$ (line 41). Finally, when `strengthen()` returns, then either $bug = \top$ (line 45), or $f(K-2) \wedge T \implies S'$ (line 36). Hence $\mathcal{I}_2$ is a valid post-condition for `strengthen()`. Also, `propagate()` is frame monotonic since it always pushes inductive lemmas forward (the order of the two statements at lines 56–57 is crucial for this). Hence, `propagate()` maintains A_1 and A_4 at all

```
58 //-- global variables
59 var (I,T,S) : problem (P)
60 var ∀i ∈ [1,n] • F_i: frame[]
61 var K: int (size of each F_i)
62 var bug: bool (CEX flag)
63
64 //-- invariants
65 ∀j ∈ [0,K−1], let
66     f(j) = ⋀_{i∈[1,n]} ⋀_{k∈[j,K−1]} ⋀_{α∈F_i[k]} α
67
68 B_1 : ∀j ∈ [0,K−1] • I ⟹ f(j)
69 B_2 : ∀j ∈ [0,K−2] • f(j) ∧ T ⟹ f′(j+1)
70 B_3 : ∀j ∈ [0,K−3] • f(j) ∧ T ⟹ S′
71 B_4 : ∀j ∈ [0,K−2] • f(j) ∧ T ⟹ S′
```

```
72 bool IC3Sync (n)
73   if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S′ ≠ ⊥)
74     return ⊥;
75   K := 3; bug := ⊥;
76   ∀i ∈ [1,n] • F_i[0] := I; F_i[1] := F_i[2] := ∅;
77   while (⊤)
78     @INV{I_4 : B_1 ∧ B_2 ∧ B_3}
79     {strengthen(F_1,K);propagate(F_1,K)}
80                 ∥ ··· ∥
81     {strengthen(F_n,K);propagate(F_n,K)};
82     @INV{I_5 : bug ∨ (B_1 ∧ B_2 ∧ B_4)}
83     if (bug) return ⊥;
84     @INV{I_6 : B_1 ∧ B_2 ∧ B_4}
85     if (∃j ∈ [1,K−2] • ∀i ∈ [1,n] • F_i[j] = ∅)
86       return ⊤;
87     @INV{I_6}
88     ∀i ∈ [1,n] • F_i[K] := ∅; K := K+1;
```

Fig. 2. Pseudo-Code for IC3SYNC. Variables are passed by reference. Functions `strengthen()` and `propagate()` are defined in Fig. 1.

times. It also maintains A_2 since a new lemma α is added to frame $F[i+1]$ (line 56) only if $f(i) \wedge T \implies \alpha'$ (line 55). Hence $\mathcal{I}_3$ is a valid post-condition for `propagate()`. Finally, note that $A_4 \equiv A_3 \wedge f[\mathbf{K}-2] \implies S$. Hence, after $\mathbf{K}$ is incremented, A_4 becomes A_3. Also, since the last frame is initialized to $\emptyset$, A_1 and A_2 are preserved. Hence: $\{\mathcal{I}_3\}\mathbf{F}[\mathbf{K}] := \emptyset; \mathbf{K} := \mathbf{K}+1; \{\mathcal{I}_1\}$. The correctness of IC3 is summarized by Theorem 1. Its proof is in Appendix A.

Theorem 1. *If* `IC3()` *returns* $\top$*, then the problem is safe. If* `IC3()` *returns* $\bot$*, then the problem is unsafe.*

We now present the three versions of parallel IC3 and their correctness (their termination follows in the same way as IC3 [5] – see Theorem 5 in Appendix A).

3.1 Synchronous Parallel IC3

The first parallelized version of IC3, denoted IC3SYNC, runs a number of copies of the sequential IC3 "synchronously" in parallel. Let IC3SYNC(n) be the instance of IC3SYNC consisting of n copies of IC3 executing concurrently. The copies maintain separate frames. However, for any copy, the frames of other copies act as "background lemmas". Specifically, the copies interact by: (i) using the frames of all other copies when computing $f(i)$; (ii) declaring the problem unsafe if any copy finds a counterexample; (iii) declaring the problem safe if some frame becomes empty across all the copies; and (iv) "synchronizing" after each call to `strengthen()` and `propagate()`.

The pseudo-code for IC3SYNC(n) is shown in Fig. 2. The main function is `IC3Sync()`. After checking the base cases (lines 73–74), it initializes flags and frames (lines 75–76), and then iteratively performs the following steps: (i) run n copies IC3 where each copy does a single step of `strengthen()` followed by `propagate()` (lines 79–81); (ii) check if any copy of IC3 found a counterexample, and if so, terminate (line 83); (iii) check if a proof of safety has been found, and if

```
//-- invariants
∀j ∈ [0, max(K_1, ..., K_n) − 1], let
    f(j) = ⋀_{i∈[1,n]} ⋀_{k∈[j,K_i−1]} ⋀_{α∈F_i[k]} α

C_1 : ∀j ∈ [0, K_i − 1] . I ⟹ f(j)
C_2 : ∀j ∈ [0, K_i − 2] . f(j) ∧ T ⟹ f′(j + 1)
C_3 : ∀j ∈ [0, K_i − 3] . f(j) ∧ T ⟹ S′
C_4 : ∀j ∈ [0, K_i − 2] . f(j) ∧ T ⟹ S′



//-- top-level function
bool IC3Async (n)
  if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S′ ≠ ⊥)
    return ⊥;
  bug := ⊥;
  IC3Copy(1) ⋄···⋄ IC3Copy(n);
  return bug ? ⊥ : ⊤;

//-- global variables
var (I, T, S) : problem (P)
var ∀i ∈ [1, n] . F_i: frame[]
var ∀i ∈ [1, n] . K_i: int (size of F_i)
var bug: bool (CEX flag)

void IC3Copy (i)
  K_i := 3; F_i[0] := I;
  F_i[1] := ∅; F_i[2] := ∅;
  while (⊤)
    @INV{I_7 : C_1 ∧ C_2 ∧ C_3}
    strengthen(F_i, K_i);
    @INV{I_8 : bug ∨ (C_1 ∧ C_2 ∧ C_4)}
    if (bug) return;
    @INV{I_9 : C_1 ∧ C_2 ∧ C_4}
    propagate(F_i, K_i);
    if (∃j ∈ [1, K_i − 2] . ∀i ∈ [1, n] . F_i[j] = ∅)
      return;
    @INV{I_9}
    F_i[K_i] := ∅; K_i := K_i + 1;
```

Fig. 3. Pseduo-Code for IC3ASYNC. Variables are passed by reference. Functions `strengthen()` and `propagate()` are defined in Fig. 1.

so, terminate (lines 85–86); and (iv) add a frame and repeat from step (i) above (line 88). Functions `strengthen()` and `propagate()` are syntactically identical to IC3 (cf. Fig. 1). However, the key semantic difference is that lemmas from all copies are used to define $f(j)$ (lines 65–66). Global variables are shared, and accessed atomically. Note that even though all IC3 copies write to variable *bug*, there is no race condition since they always write the same value ($\top$).

Correctness. The correctness of IC3SYNC follows from the invariants specified in Fig. 2. To show these invariants are valid, the main challenge is to show that if $\mathcal{I}_4$ holds at line 78, then $\mathcal{I}_5$ holds at line 82. Note that since `strengthen()` and `propagate()` are frame monotonic, they preserve B_1 and B_3. This means that $B_1 \wedge B_3$ holds at line 82. Now suppose that at line 82, we have $\neg bug$. This means that each `strengthen()` called at lines 79–81 returned from line 36. Thus, the condition $f(\mathbf{K}-2) \wedge T \implies S'$ was established at some point, and once established, it continues to hold due to the frame monotonicity of `strengthen()` and `propagate()`. Since $B_4 \equiv B_3 \wedge (f(\mathbf{K}-2) \wedge T \implies S')$, we therefore know that $B_1 \wedge B_4$ holds at line 82. Also, B_2 holds at line 82 since a new lemma α is only added to frame $F_i[j+1]$ by `strengthen()` (line 42) and `propagate()` (line 56) under the condition $f(j) \wedge T \implies \alpha'$. Note that once $f(j) \wedge T \implies \alpha'$ is true, it continues to hold even in the concurrent setting due to frame monotonicity. Finally, the statement at line 88 transforms $\mathcal{I}_6$ to $\mathcal{I}_4$. The correctness of IC3SYNC is summarized by Theorem 2. Its proof is in Appendix A.

Theorem 2. *If* `IC3Sync()` *returns* $\top$*, then the problem is safe. If* `IC3Sync()` *returns* $\bot$*, then the problem is unsafe.*

3.2 Asynchronous Parallel IC3

The next parallelized version of IC3, denoted IC3ASYNC, runs a number of copies of the sequential IC3 "asynchronously" in parallel. Let IC3ASYNC(n) be the in-

```
//-- global variables
var (I,T,S) : problem (P)
var ∀i ∈ [1,n].F_i, P_i: frame[]
var ∀i ∈ [1,n].K_i: int (size of F_i and P_i)
var bug, safe: bool (CEX and proof flags)


void IC3PrCopy (i)
  K_i := 3; F_i[0] := I;
  F_i[1] := ∅; F_i[2] := ∅;
  while (⊤)
    @INV{I_7 : C_1 ∧ C_2 ∧ C_3}
    strengthen(F_i, K_i);
    @INV{I_8 : bug ∨ (C_1 ∧ C_2 ∧ C_4)}
    if (bug) return;
    @INV{I_9 : C_1 ∧ C_2 ∧ C_4}
    propProof(F_i, K_i);
    if (safe) return;
    @INV{I_9}
    F_i[K_i] := ∅; K_i := K_i + 1;

bool IC3Proof (n)
  if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S' ≠ ⊥)
    return ⊥;
  bug := ⊥; safe := ⊥;
  IC3PrCopy(1) ◇···◇ IC3PrCopy(n);
  return bug ? ⊥ : ⊤;

void propProof(F, K)
  for j : 1 ... K − 2
    for α ∈ F[j]
      if (f(j) ∧ T ⟹ α')
        F[j + 1] := F[j + 1] ∪ {α};
        F[j] := F[j] \ {α};
        if (F[j] = ∅)
          P_j := ⋃_{j<k≤K−1} F[k];
          Π := ⋃_{{i|1≤i≤n∧j<K_i}} P_i;
          if (Π ∧ T ⟹ Π')
            safe := ⊤; return;
```

Fig. 4. Pseduo-Code for IC3PROOF. Variables are passed by reference. Function `strengthen()` is defined in Fig. 1. Formulas $f(j), \mathcal{I}_7, \mathcal{I}_8$, and $\mathcal{I}_9$ are defined in Fig. 3.

stance of IC3ASYNC consisting of n copies of IC3 executing concurrently. Similar to IC3SYNC, the copies maintain separate frames, interact by sharing lemmas when computing $f(i)$, and declare the problem unsafe if any copy finds a counterexample. However, due to the lack of synchronization, proof detection is distributed over all the copies instead of being centralized in the main thread.

Fig. 3 shows the pseudo-code for IC3ASYNC(n). The main function is `IC3Async()`. After checking the base cases (lines 102–103), it initializes flags (line 104), lauches n copies of IC3 in parallel (line 105) and waits for some copy to terminate (the $\diamond$ operator), and checks the flag and returns with an appropriate result (line 106). Function `IC3Copy()` is similar to `IC3()` in Fig. 1. The key difference is that lemmas from all copies are used to compute $f(j)$ (lines 90–91).

Correctness. The correctness of IC3ASYNC follows from the invariants specified in Fig. 3. To see why these invariants are valid, note that C_1 and C_3 are always preserved due to frame monotonicity. If `strengthen()` returns with $bug = \bot$, then it returned from line 36, and hence $f(\mathbf{K}_i - 2) \wedge T \implies S'$ was true at some point in the past and continues to hold due to frame monotonicity. Together with C_3, this implies that C_4 holds at line 119. Also, C_2 holds at line 119 since a new lemma α is only added to frame $F_i[j+1]$ by `strengthen()` (line 42) and `propagate()` (line 56) under the condition $f(j) \wedge T \implies \alpha'$. Note that once $f(j) \wedge T \implies \alpha'$ is true, it continues to hold even under concurrency due to frame monotonicity. Hence, $\mathcal{I}_8$ holds at line 119. Since *bug* is never set to $\bot$, this means that $\mathcal{I}_9$ holds at line 121 even under concurrency. Finally, the statement at line 126 transforms $\mathcal{I}_9$ to $\mathcal{I}_7$. The correctness of IC3ASYNC is summarized by Theorem 3. Its proof is in Appendix A.

Theorem 3. *If* `IC3Async()` *returns* $\top$*, then the problem is safe. If* `IC3Async()` *returns* $\bot$*, then the problem is unsafe.*

3.3 Asynchronous Parallel IC3 With Proof-Checking

The final parallelized version of IC3, denoted IC3PROOF, is similar to IC3ASYNC, but add more aggressive checking for proofs. Let IC3PROOF(n) be the instance of IC3ASYNC consisting of n copies of IC3 executing concurrently. Similar to IC3ASYNC, the copies maintain separate frames, interact by sharing lemmas when computing $f(i)$, and declare the problem unsafe if any copy finds a counterexample. However, whenever a copy finds an empty frame, it checks whether the set of lemmas over all the copies for the frame forms an inductive invariant.

The pseudo-code for IC3PROOF(n) is shown in Fig. 4. The main function is `IC3Proof()`. After checking the base cases (lines 148–149), it initializes flags (line 150), lauches n copies of IC3 in parallel (line 151) and waits for at least one copy to terminate, and checks the flag and returns with an appropriate result (line 152). Each copy of IC3 is similar to the sequential IC3 in Fig. 1. The key difference is in `propProof()` where, once an empty frame is detected (line 160), we check whether a proof has been found by collecting the lemmas for the frame (lines 161–162), and checking if these lemmas are inductive (line 163).

Correctness. The correctness of IC3PROOF follows from the invariants (whose validity is similar to those for IC3ASYNC) specified in Fig. 4. It is summarized by Theorem 4. The proof of the theorem is in Appendix A.

Theorem 4. *If* `IC3Proof()` *returns* $\top$*, then the problem is safe. If* `IC3Proof()` *returns* $\bot$*, then the problem is unsafe.*

4 Parallel IC3 Portfolios

In this section, we investigate the question of how a good portfolio size can be selected if we want to implement a portfolio of IC3PARs. We begin with an argument about the pdf of the runtime of IC3ASYNC(n).

Conjecture 1. The runtime of IC3ASYNC(n) converges to a Weibull rv as $n \to \infty$.

Argument: Recall that each execution of IC3ASYNC(n) consists of n copies of IC3 running in parallel, and that IC3ASYNC(n) stops as soon as one copy finds an answer. We can consider the runtime of each copy of IC3 to be a rv. Specifically, let X_i be the rv denoting the runtime of the i-th copy of IC3 assuming it was allowed to run till completion. Recall that the pdf of X_i has a lower bound of 0, since no run of IC3 can take negative time. Also the set of random variables $(X_1, \dots, X_n)$ are i.i.d. since the copies of IC3 only interact with each other logically. Finally, let X be the random variable denoting the runtime of IC3ASYNC(n). Note that $X = \min(X_1, \dots, X_n)$. Hence, by the EVT, $X \sim \text{WEI}(k, \lambda)$ for large n. □

A similar argument holds for IC3SYNC and IC3PROOF, and therefore their runtime should follow Weibull as well. In the rest of this section, we write IC3PAR to mean a specific parallel IC3 variant. Empirically, we find that the runtime of IC3PAR(n) follows a Weibull distribution closely for even modest values of n. Specifically, we selected 10 examples (5 safe and 5 buggy) from HWMCC14, and

	IC3SYNC (4)				IC3ASYNC (4)				IC3PROOF (4)			
Example	k	λ	μ, μ^*	σ, σ^*	k	λ	μ, μ^*	σ, σ^*	k	λ	μ, μ^*	σ, σ^*
6s286	4.07	1119	1015,1015	280,274	4.44	990	902,903	230,220	4.35	980	892,892	232,228
intel026	2.71	49.0	43.6,44.2	17.3,14.6	3.70	50.2	45.3,46.2	13.6,10.1	3.70	50.1	45.2,46.1	13.6,10.3
6s273	3.80	26.1	23.6,23.6	6.93,6.57	4.11	23.5	21.3,21.4	5.85,5.36	4.17	23.3	21.2,21.3	5.73,5.29
intel057	6.58	16.0	14.9,15.1	2.66,2.11	7.31	17.2	16.1,16.1	2.60,2.46	7.52	17.8	16.7,16.9	2.63,2.07
intel054	7.82	24.3	22.8,23.0	3.46,2.94	8.69	26.1	24.6,24.8	3.38,2.84	9.26	26.1	24.7,24.8	3.20,2.92
6s215	2.38	7.69	6.82,7.03	3.05,2.34	4.71	6.75	6.17,6.21	1.49,1.34	4.72	6.38	5.84,5.90	1.41,1.21
6s216	1.95	35.1	31.1,31.0	16.6,16.9	3.56	27.5	24.8,24.9	7.74,6.97	2.78	28.1	25.0,25.1	9.74,9.05
oski3ub1i	5.98	54.9	50.9,51.4	9.90,7.90	7.02	52.3	48.9,49.2	8.20,6.71	4.78	54.8	50.2,50.8	11.9,9.53
oski3ub3i	5.71	52.4	48.5,48.9	9.84,8.00	5.51	52.2	48.2,48.6	10.1,8.51	5.66	52.2	48.2,48.5	9.87,8.39
oski3ub5i	5.08	66.8	61.4,61.9	13.8,11.6	4.94	67.2	61.6,62.0	14.2,12.4	4.93	66.2	60.7,61.1	14.0,12.1
SAFE	5.00	246	224,224	62.1,60.2	5.65	221	202,202	51.1,48.3	5.80	219	200,200	51.4,49.7
BUG	4.22	43.4	39.7,40.0	10.6,9.37	5.15	41.2	37.9,38.2	8.36,7.20	4.58	41.5	38.0,38.3	9.42,8.07
ALL	4.61	145	131,132	36.4,34.7	5.40	131	120,120	29.7,27.7	5.19	130	119,119	30.4,28.9

Fig. 5. Fitting IC3PAR(4) runtime to Weibull. First 5 examples are safe, next 5 are buggy; SAFE, BUG, ALL = average over safe, buggy, and all examples; μ, μ^* = predicted, observed mean; σ, σ^* = predicted, observed standard deviation.

for each example we: (i) executed IC3ASYNC(4) around 3000 times; (ii) measured the runtimes; (iii) estimated the k and λ values for the Weibull distribution that best fits these values; and (iv) computed the observed mean and standard deviation from the data, and the predicted mean and standard deviation from the k and λ estimates. We repeated these experiments with IC3SYNC and IC3PROOF.

The results are shown in Fig. 5(a). We see that in all cases, the observed mean and standard deviation is quite close to the predicted ones, indicating that the estimated Weibull distribution is a good fit for the measured runtimes. IC3ASYNC and IC3PROOF have similar performance, are and slightly faster overall than IC3SYNC, indicating that additional synchronization is counter-productive. The estimated k and λ values vary widely over the examples, indicating their diversity. Note that smaller values of λ mean a smaller expected runtime.

Determining Portfolio Size. Consider a portfolio of IC3PARs. In general, increasing the size of the portfolio reduces the expected time to solve a problem. However, there is diminishing returns to adding more solvers to a portfolio in terms of expected runtime. We now express this mathematically, and derive a formula for computing a portfolio size to achieve an runtime with a target probability. Consider a portfolio of m IC3PAR solvers run on a specific problem. Let Y_i denote the runtime of the i-th IC3PAR. From previous discussion we know that $Y_i \sim \textsc{wei}(k, \lambda)$ for some k and λ. Therefore, the cdf of Y_i is: $F_{Y_i}(x) = 1 - e^{-(\frac{x}{\lambda})^k}$.

Let Y be the rv denoting the runtime of the portfolio. Thus, we have $Y = \min(Y_1, \ldots, Y_m)$. More importantly, the cdf of Y is:

$$\begin{aligned} F_Y(x) &= 1 - (1 - F_{Y_1}(x)) \times \cdots \times (1 - F_{Y_m}(x)) \\ &= 1 - (e^{-(\frac{x}{\lambda})^k})^m = 1 - e^{-m(\frac{x}{\lambda})^k} = 1 - e^{-(\frac{xm^{\frac{1}{k}}}{\lambda})^k} \end{aligned}$$

Note that this means Y is also a Weibull rv, not just when $m \to \infty$ (as per the EVT) but for all m. More specifically, $Y \sim \textsc{wei}(k, \frac{\lambda}{m^{\frac{1}{k}}})$. Recall that if $m = 1$, then the expected time to solve the problem by the portfolio is $E[Y_1]$.

We refer to this time as t^*, the expected solving time for a single IC3PAR. Since $Y_1 \sim \textsc{wei}(k, \lambda)$, it is known that $t^* = \lambda\Gamma(1+\frac{1}{k})$, where Γ is the gamma function. Now, we come to our result, which expresses the probability that a portfolio of m IC3PARs will require no more than t^* to solve the problem.

Theorem 5. *Let $p(m)$ be the probability that $Y \leq t^*$. Then $p(m) > 1 - e^{-\frac{m}{e^\gamma}}$ where $\gamma \approx 0.57721$ is the Euler-Mascheroni constant.*

Proof. We know that:

$$p(m) = F_Y(t^*) = 1 - e^{-m(\Gamma(1+\frac{1}{k}))^k} = 1 - (\alpha(k))^m, \text{ where } \alpha(k) = e^{-(\Gamma(1+\frac{1}{k}))^k}$$

Next, observe that $\alpha(k)$ increases monotonically with k but does not diverge as $k \to \infty$. For example, Fig. 11 in Appendix B shows a plot of $\alpha(k)$. Indeed, it can be shown that (see Lemma 2 in Appendix B): $\lim_{k\to\infty} \alpha(k) = e^{-\frac{1}{e^\gamma}}$. In practice, as seen in Fig. 11 in Appendix B, the value of $\alpha(k)$ converges quite rapidly to this limit as k increases. For example, $\alpha(5) > 0.91 \cdot e^{-\frac{1}{e^\gamma}}$, and $\alpha(10) > 0.95 \cdot e^{-\frac{1}{e^\gamma}}$. Since $\forall k \centerdot \alpha(k) < e^{-\frac{1}{e^\gamma}}$, we have our result:

$$p(m) > 1 - (e^{-\frac{1}{e^\gamma}})^m = 1 - e^{-\frac{m}{e^\gamma}}$$

Achieving a Target Probability. Now suppose we want p_m to be greater than some target probability p. Then, from Theorem 5, we have:

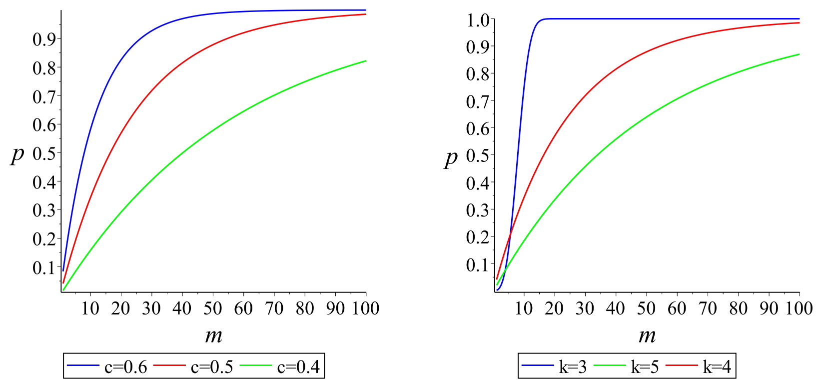
$$p = 1 - (e^{-\frac{1}{e^\gamma}})^m \iff 1 - p = e^{-\frac{m}{e^\gamma}} \iff \ln(1-p) = -\frac{m}{e^\gamma}$$
$$\iff \ln(\tfrac{1}{1-p}) = \frac{m}{e^\gamma} \iff m = e^\gamma \ln(\tfrac{1}{1-p})$$

For example, if we want $p = 0.99999$, then $m \approx 20$. Thus, a portfolio of 20 IC3PARs has about 0.99999 probability of solving a problem at least as quickly as the expected time in which a single IC3PAR will solve it. We validated the efficacy of Theorem 5 by comparing its predictions with empirically observed results on the HWMCC14 benchmarks. Overall, we find the observed and predicted probabilities to agree significantly. Further details are presented in Section 5.2.

Speeding Up the Portfolio. To reduce the portfolio's runtime below t^*, we must increase m appropriately. In general, for any constant $c \in [0, 1]$, the probability that a portfolio of m IC3PAR solvers will have a runtime $\leq c \cdot t^*$ is given by:

$$p(m, c, k) = 1 - e^{-m(c \cdot \Gamma(1+\frac{1}{k}))^k}$$

For $c < 1$ we do not have a closed form for $\lim\limits_{k\to\infty} p(m, c, k)$, unlike when $c = 1$. However, the value of $p(m, c, k)$ is computable for fixed m, c and k. Fig. 6(a) plots $p(m, c, 4)$ for $m = \{1, \ldots, 100\}$ and $c = \{0.4, 0.5, 0.6\}$. Fig. 6(b) plots $p(m, .5, k)$ for $m = \{1, \ldots, 100\}$ and $k = \{3, 4, 5\}$. As expected, $p(m, c, k)$ increases with: (i) increasing m; (ii) increasing c; and (iii) decreasing k. One challenge here is that we do not know how to estimate k for a problem without actually solving it. In general, a smaller value of k means that a smaller portfolio will reach the target probability. In our exeriments – recall Fig. 5(a) – we observed k-values in a small range (1–10) for problems from HWMCC14. These numbers can serve as guidelines, and could be refined based on additional experimentation.



(a) (b)

Fig. 6. (a) $p(m, c, 4)$ for different values of c; (b) $p(m, .5, k)$ for different values of k.

5 Experimental Results

We implemented IC3SYNC, IC3ASYNC and IC3PROOF by modifying a publicly available reference implementation of IC3 (`https://github.com/arbrad/IC3ref`), which we call IC3REF. All propositional queries in IC3 are implemented by calls to MINISAT. We refer to the variant of IC3REF that uses a randomized MINISAT (invoked via `IC3 -r`) as IC3RND. We use IC3RND to introduce uncertainty in the proof search by IC3 purely by randomizing the backend SAT solver. We performed two sets of experiments – one to evaluate the effectivess of the parallel IC3 solvers, and another to validate our statistical analysis of their portfolios. All our tools and results are available at `http://somewhere`.

Benchmarks. We constructed four benchmarks. The first was constructed by taking the safe examples from HWMCC14 (`http://fmv.jku.at/hwmcc14cav`), simplifying them with the IIMC (`http://ecee.colorado.edu/wpmu/iimc`) tool (via `iimc-hwmcc13 -t pp`), and selecting the ones solved by IC3REF within 900s on a 8 core 3.4GHz machine with 8GB of RAM. The remaining three benchmarks were constructed similarly from the buggy examples from HWMCC14, and the safe and buggy examples from the TIP benchmark (`http://fmv.jku.at/aiger/tip-aig-20061215.zip`), respectively. We refer to the four benchmarks as HWCSAFE, HWCBUG, TIPSAFE, and TIPBUG, respectively.

SAT Solver Pool. The function f (cf. Figs. 1–4) is implemented by a SAT solver (MINISAT). A separate SAT solver S_i is used for each $f(i)$. Whenever $f(i)$ changes due to the addition of a new lemma to a frame, the corresponding solver S_i is also updated by asserting the lemma. To avoid a single SAT solver from becoming the bottleneck between competing threads, we use a "pool" of MINISAT solvers to implement each S_i. The solvers are maintained in a FIFO queue. When a thread requests a solver, the first available solver is given to it. When a lemma is added to the pool, it is added to all available solvers, and recorded as "pending" for the busy ones. When a busy solver is returned by a

		IC3SYNC		IC3ASYNC		IC3PROOF		IC3RND	
$\mathcal{B}$	$\mathcal{B}^*$	Mean	Max	Mean	Max	Mean	Max	Mean	Max
HWCSAFE	31	1.30	5.61	1.58	5.47	1.60	4.08	1.17	4.64
HWCBUG	14	2.49	18.7	14.3	151	25.1	309	1.07	1.49
TIPSAFE	14	1.28	4.50	2.61	11.1	2.29	12.8	1.37	3.80
TIPBUG	9	2.23	5.35	2.82	7.32	3.50	12.1	1.16	2.17
SAFE	44	1.30	5.61	1.93	11.1	1.83	12.8	1.24	4.64
BUG	23	2.38	18.7	9.58	151	16.3	309	1.11	2.17
ALL	67	1.67	18.7	4.74	151	6.79	309	1.19	4.64

Fig. 7. Speedup of IC3SYNC, IC3ASYNC, IC3PROOF and IC3RND compared to IC3REF.

thread, all pending lemmas are added, and the solver is inserted at the back of the queue. We refer to the number of solvers in each pool as $SPSz$.

5.1 Comparing Parallel IC3 Variants

These experiments were carried on a Intel Xeon machine with 128 cores, each running at 2.67GHz, and 1TB of RAM. For each solver S selected from $\{$IC3ASYNC(4), IC3SYNC(4), IC3PROOF(4), IC3RND$\}$ and each benchmark $\mathcal{B}$, and with $SPSz = 3$, we performed the following steps: (i) extract all problems from $\mathcal{B}$ that are solved by IC3REF in at least 10s; call this set $\mathcal{B}^*$; (ii) solve each problem in $\mathcal{B}^*$ with IC3REF and also with a portfolio of 20 S solvers, compute the ratio of the two runtimes; this is the speedup; (iii) compute the mean and max of the speedups over all problems in $\mathcal{B}^*$. Figure 7 shows the results obtained. In all cases, we see speedup. On this particular run, IC3PROOF performs best overall, with an average speedup of over 6 and a maximum speedup of over 300. Note however, that performance will vary across runs due to unpredictability of runtime. As in the non-portfolio case (cf. Fig. 5) IC3PROOF and IC3ASYNC have similar performance, and are better than IC3SYNC. The pattern is followed for both safe and buggy examples. Finally, IC3RND provides mediocre speedup across all examples (cf. the "Max" column) indicating that parallelization enables broader search for proofs compared to randomizing the SAT solver.

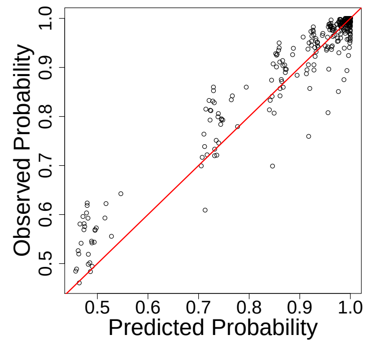
5.2 Portfolio Size

To validate Theorem 5, we compared its predictions to empirically observed results as follows (again using $SPSz = 3$):

1. Select a set of problems – same as in Fig. 5(a) – from HWMCC14, and process each problem as follows.
2. Solve the problem b times using IC3PAR(4). These experiments are the same as the ones used for Fig. 5(a). Hence b is the value appearing in the second column of Fig. 5(a). This gives a set of runtimes $t_1, \ldots, t_b$. Fit these runtimes to a Weibull distribution to obtain the estimated value of k (the same as the third column of Fig. 5(a)).
3. Compute $\tilde{t} = \text{mean}(t_1, \ldots, t_b)$. This is the estimated average time for IC3PAR(4) to solve the problem.

	ρ - IC3ASYNC		ρ - IC3SYNC		ρ - IC3PROOF	
Example	Mean	StDev	Mean	StDev	Mean	StDev
6s286	1.0000	0.0016	1.0010	0.0046	0.9996	0.0032
intel026	1.0042	0.0233	1.0028	0.0163	1.0027	0.0163
6s273	1.0025	0.0122	1.0031	0.0149	1.0030	0.0154
intel057	0.9968	0.0214	0.9855	0.0381	1.0002	0.0136
intel054	1.0029	0.0162	0.9998	0.0076	0.9994	0.0080
6s215	1.0001	0.0057	0.9988	0.0099	0.9991	0.0058
6s216	1.0038	0.0204	1.0025	0.0163	1.0034	0.0182
oski3ub1i	1.0063	0.0321	1.0055	0.0293	1.0049	0.0274
oski3ub3i	1.0042	0.0230	1.0049	0.0259	1.0053	0.0272
oski3ub5i	1.0061	0.0312	1.0070	0.0358	1.0069	0.0357

(a)



(b)

Fig. 8. Validating Theorem 5; (a) mean and standard deviation of ratios of predicted and observed probabilities; (b) scatter plot of predicted and observed probabilities.

4. Pick a portfolio size m. Start with $m = 1$.
5. Divide $t_1, \ldots, t_b$ into blocks of size m. Let $B = \lfloor \frac{b}{m} \rfloor$. We now have B blocks of runtime $T_1, \ldots, T_B$, each consisting of m elements. Thus, $T_1 = \{t_1, \ldots, t_m\}$, $T_2 = \{t_{m+1}, \ldots, t_{2m}\}$, and so on. For $i = 1, \ldots, B$, compute $\mu_i = \min(T_i)$. Note that each μ_i represents the runtime of a portfolio of m IC3PAR(4) solvers on the problem.
6. Let $n(m)$ be the number of blocks for which $\mu_i \leq \tilde{t}$, i.e., $n(m) = |\{i \in [1, B] \mid \mu_i \leq \tilde{t}\}|$. Compute $p^*(m) = \frac{n(m)}{B}$. Note that $p^*(m)$ is the estimate of $p(m)$ based on our experiments. Compute $p(m) = 1 - (\alpha(k))^m$ using the estimated value of k from Step 2. Compute $\rho(m) = \frac{p^*(m)}{p(m)}$. We expect $\rho(m) \approx 1$.
7. Repeat steps 5 and 6 with $m = 2, \ldots, 100$ to obtain the sequence $\rho = \langle \rho(1), \ldots, \rho(100) \rangle$. Compute the mean and standard deviation of ρ.

Fig. 8(a) shows the results of the above steps over all the selected examples. We see that for each example, the mean of ρ is very close to 1 and its standard deviation is very close to 0, indicating that $p(m)$ and $p^*(m)$ agree considerably. Furthermore, Fig. 8(b) shows a scatter plot of all $p^*(m)$ values computed against their corresponding $p(m)$. Note that most values are very close to the (red) $x = y$ line, as expected.

5.3 Parameter Sweeping

In this section we evaluate the performance of IC3PROOF when selecting diffmuch combinations of IC3PAR parameters. We observed in 5.1 that the variants of IC3PAR each have a chance of being the best solver for diffrent benchmarks. From the previous work utilizing the portfolio technique (TODO prune this list) [20, 17, 12, 14, 19, 16], we see that using a suite of heterogeneous solvers would

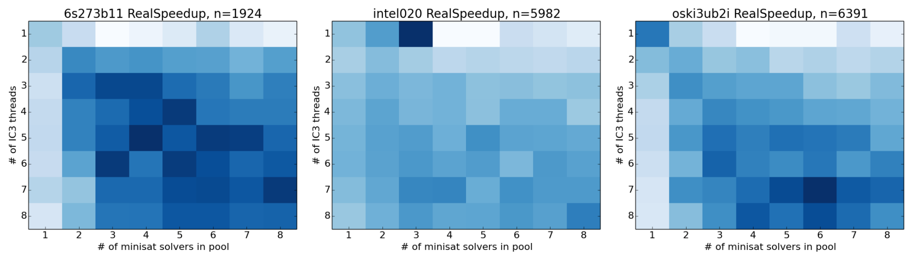


Fig. 9. IC3PROOF speedup on three benchmarks compared to IC3REF. The intensity of a cell indicates the corresponding combination of IC3PAR parameters solves the benchmark faster.

likely be successfull. With this inspiration, we determined that running portfolios of IC3PAR in differnt configurations could be successfull. This effort would also help to better characterize the behavior of IC3PAR.

IC3PAR has two parameters: number of threads running a copy of IC3, and $SPSz$. We identify an instance of IC3PAR run with these parameters as IC3PAR(i, s) where i is the number of IC3 threads and $SPSz = s$. Thus, IC3PROOF$(4, 3)$ was used is all previous experiments.

Conjecture 2. In the abscence of knowledge of optimal parameter values for IC3PAR(i, s), a heterogeneous portfolio using random feasible parameter values (I, S) will, on average, yield faster IC3PAR performance than a homogenous portfolio with constant parameter values $(i, s) = (4, 3)$. Where I and S are defined by a random discrete variable $r.v.\ X = x; \quad x \in \{1, 2, \ldots, 8\}$.

To investigate Conjecture 2, we estimated the speedup over IC3 for portfolios of IC3PROOF(i, s) and IC3PROOF(I, S) as follows:

1. Select a benchmark from $\mathcal{B}$ and time its performance with IC3
2. Time 100 runs if IC3PROOF in each of the 64 possible parameter combinations IC3PROOF(I, S) to empiracly charachterize the random running time distribution across the parameter space.
3. Select randomly 100 portfolio blocks consisting of 20 run times of IC3PROOF(i, s) from the 6,400 recorded running times convering IC3PROOF(I, S) performed in Step 2 Take minimum of each block as the portfolio time, and average the 100 minimums.
4. Select the 5 portfolio blocks of size 20 from the 100 runs of IC3PROOF$(4, 3)$ which were performed as part of Step 2. Take minimum of each block as the portfolio time, and average the 5 minimums.

For this investigation we constructed a portfolio simulator which used the running times gathered from up to 6400 tests per benchmark and constructed ex post facto portfolios by selecting running times from the desired parameter configuration. We utilized over 11,000 hours of compute time across 11 dual processor machines with Intel(R) Xeon(R) 2.40GHz CPU's for a total of 176 cores.

	Time	Speedup					Time	Speedup			
Example	IC3REF	Sync	Async	Proof	Sweep	Example	IC3REF	Sync	Async	Proof	Sweep
6s286	947.6	1.70	1.75	1.88	1.78	6s215	12.20	2.79	3.11	3.20	2.35
intel026	78.33	2.63	2.36	2.37	2.58	6s216	67.24	4.66	4.41	4.90	4.22
6s273	31.06	2.22	2.30	2.28	1.63	oski3ub1i	83.64	2.06	2.12	2.22	1.96
intel057	31.33	2.54	2.40	2.30	2.64	oski3ub3i	79.41	2.12	2.23	2.18	1.95
intel054	55.89	3.08	2.78	2.79	3.93	oski3ub5i	127.3	2.81	2.87	2.87	2.79

Fig. 10. Speedup via parameter sweeping.

Summarizing graphics were produced for visualization of performance across the parameter space (see Fig. 9.) The visualizations and simulated results presented evidence in favor of 2, as speedup patterns across the parameter space were varied for all of the selected benchmark examples and every simulated IC3PROOF(I, S) portfolio ran faster than simulated IC3PROOF$(4, 3)$ portfolios.

To attempt to validate the conjecture, actual heterogenous IC3PAR(I, S) portfolios were run on the 10 selected benchmarks from Section 4. Each portfolio was run at least XXX times, and the average portfolio times were then compared. These results (shown in Figure 10) show that averaged across these 10 examples, IC3PAR(I, S) is as fast as any single IC3PAR variant. The limited ammount of data collected to validate this conjecture does not support any strong claims, but from what we have observed: using heterogenous portfolios of IC3PROOF gives the same speedup as picking the best possible IC3PAR variant. The advantage to this technique is for a new problem when the strongest performing variant can not be known ahead of time.

6 Conclusion

We present three ways to parallelize IC3. Each variant uses a number of threads to speed up the computation of an inductive invariant or a CEX, sharing lemmas to minimize duplicated effort. They differ in the degree of synchronization and technique to detect if an inductive invariant has been found. The runtime of these solvers is unpredictable, and varies with thread-interleaving. We explore the use of portfolios to counteract the runtime variance. Each solver in the portfolio potentially searches for a different proof/CEX. The first one to succeed provides the solution. Using the Extreme Value theorem and statistical analysis, we construct a formula that gives us a portfolio size to solving a problem within a target time bound with a certain probability. Experiments on HWMCC14 benchmarks show that the combination of parallelization and portfolios yields an average speedups of 6x over IC3, and in some cases speedups of over 300. An important area of future work is the effectiveness of parallelization and portfolios in the context of software verification via a generalization of IC3 [10].

Acknowledgment. We are grateful to Jeffery Hansen and Arie Gurfinkel for helpful comments and discussions[1].

[1] This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002110

References

1. Albarghouthi, A., Kumar, R., Nori, A.V., Rajamani, S.K.: Parallelizing top-down interprocedural analyses. In: Vitek, J., Lin, H., Tip, F. (eds.) Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12). pp. 217–228. Association for Computing Machinery, Beijing, China (June 2012)
2. Barnat, J., Brim, L., Havel, V., Havlícek, J., Kriho, J., Lenco, M., Rockai, P., Still, V., Weiser, J.: DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: Sharygina, N., Veith, H. (eds.) Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13). Lecture Notes in Computer Science, vol. 8044, pp. 863–868. Springer-Verlag, Saint Petersburg, Russia (July 2013)
3. Bingham, B., Bingham, J., Erickson, J., de Paula, F.M., Reitblatt, M., Singh, G.: Industrial Strength Distributed Explicit State Model Checking. In: Proceedings of the 9th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'10). Twente, The Netherlands (September–October 2010)
4. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) Proceedings of the 22nd International Conference on Computer Aided Verification (CAV '10). Lecture Notes in Computer Science, vol. 6174, pp. 354–359. Springer-Verlag, Edinburgh, UK (July 2010)
5. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '11). Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer-Verlag, New York (January 2011)
6. Cimatti, A., Griggio, A.: Software Model Checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12). Lecture Notes in Computer Science, vol. 7358, pp. 277–293. Springer-Verlag, Berkeley, CA, USA, July 7-13, 2012. New York, NY (July 2012)
7. Ditter, A., Ceska, M., Lüttgen, G.: On Parallel Software Verification Using Boolean Equation Systems. In: Proceedings of the 19th International SPIN Workshop on Model Checking of Software (SPIN '12). pp. 80–97. Oxford, UK (July 2012)
8. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD '11). pp. 125–134. IEEE Computer Society, Austin, TX (October–November 2011)
9. de Haan, L., Ferreira, A.: Extreme Value Theory: An Introduction. Springer (2006)
10. Hoder, K., Bjørner, N.: Generalized Property Directed Reachability. In: Cimatti, A., Sebastiani, R. (eds.) Proceedings of the 15th International Conference on the Theory and Applications of Satisfiability Testing (SAT '12). Lecture Notes in Computer Science, vol. 7317, pp. 157–171. Springer-Verlag, Trento, Italy (June 2012)
11. Holzmann, G.J.: Parallelizing the Spin Model Checker. In: Proceedings of the 19th International SPIN Workshop on Model Checking of Software (SPIN '12). pp. 155–171. Oxford, UK (July 2012)
12. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm Selection and Scheduling. In: Lee, J.H. (ed.) Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11). Lecture Notes in Computer Science, vol. 6876, pp. 454–469. Springer-Verlag, Perugia, Italy (September 2011)

13. Lopes, N.P., Rybalchenko, A.: Distributed and Predictable Software Model Checking. In: Jhala, R., Schmidt, D.A. (eds.) Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '11). Lecture Notes in Computer Science, vol. 6538, pp. 340–355. Springer-Verlag, New York (January 2011)
14. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction. In: Nicosia, G., Pardalos, P.M. (eds.) Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION'13). Lecture Notes in Computer Science, vol. 7997, pp. 153–167. Springer-Verlag, Catania, Italy (January 2013)
15. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and Distributed Model Checking in Eddy. In: Valmari, A. (ed.) Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN '06). Lecture Notes in Computer Science, vol. 3925, pp. 108–125. Springer-Verlag, Vienna, Austria (March 2006)
16. Palikareva, H., Cadar, C.: Multi-solver Support in Symbolic Execution. In: Sharygina, N., Veith, H. (eds.) Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13). Lecture Notes in Computer Science, vol. 8044, pp. 53–68. Springer-Verlag, Saint Petersburg, Russia (July 2013)
17. Ppfolio website, `http://www.cril.univ-artois.fr/~roussel/ppfolio`.
18. Weibull, W.: A Statistical Distribution Function of Wide Applicability. ASME Journal of Applied Mechanics 18(3), 293–297 (September 1951)
19. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: A Concurrent Portfolio Approach to SMT Solving. In: Bouajjani, A., Maler, O. (eds.) Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09). Lecture Notes in Computer Science, vol. 5643, pp. 715–720. Springer-Verlag, Grenoble, France, June 26 - July 2, 2009. New York, NY (July 2009)
20. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research (JAIR) 32, 565–606 (2008)

A Proof of Correctness of IC3 and its Parallel Versions

We begin with a useful Lemma.

Lemma 1. *Suppose there exists an index i such that the following hold:*

$$\alpha_1 : I \implies f(i) \quad \alpha_2 : f(i) \wedge T \implies f'(i+1)$$
$$\alpha_3 : f(i) \wedge T \implies S' \quad \alpha_4 : f(i) = f(i+1) \quad \alpha_5 : I \implies S$$

Then $Post^{0+}(I) \subseteq S$.

Proof. Since $Post(\cdot)$ is monotonic:

$$\begin{aligned} \alpha_1 \wedge \alpha_2 \wedge \alpha_4 &\implies Post(I) \subseteq Post(f(i)) \\ &\implies Post(I) \subseteq f(i+1) \\ &\implies Post(I) \subseteq f(i) \end{aligned}$$

From this, applying $Post(\cdot)$ again, we get:

$$\begin{aligned} \alpha_1 \wedge \alpha_2 \wedge \alpha_4 &\implies Post^2(I) \subseteq Post(f(i)) \\ &\implies Post^2(I) \subseteq f(i+1) \\ &\implies Post^2(I) \subseteq f(i) \end{aligned}$$

Since, we can continue arbitrarily many times like this, we have:

$$\alpha_1 \wedge \alpha_2 \wedge \alpha_4 \implies Post^{1+}(I) \subseteq Post(f(i))$$

From the above and α_3, we have $Post^{1+}(I) \subseteq S$. Also, from α_5, we know that $Post^0(I) \subseteq S$. Hence, $Post^{0+}(I) \subseteq S$, which is what we want. □

We now prove the theorems in Section 3.

Theorem 1. *If* `IC3()` *returns $\top$, then the problem is safe. If* `IC3()` *returns $\bot$, then the problem is unsafe.*

Proof. If `IC3()` returns $\bot$, `strengthen()` sets *bug* to $\top$. Hence, there exists a sequence $\langle (m_0, 0), (m_1, 1), \ldots, (m_{\mathbf{K}-2}, \mathbf{K}-2) \rangle$ such that:

$$m_0 \models I \wedge m_{\mathbf{K}-2} \models \neg S \wedge \forall i \in [0, \mathbf{K}-2) \centerdot m_i \wedge T \wedge m'_{i+1} \neq \bot \qquad (1)$$

This sequence leads to a counterexample. The problem is unsafe. If `IC3()` returns $\top$, from lines 27–28 we have:

$$\exists i \in [1, \mathbf{K}-2] \centerdot \mathbf{F}[i] = \emptyset \implies \exists i \in [1, \mathbf{K}-2] \centerdot f(i) = f(i+1)$$

This, together with $\mathcal{I}_3$, the check for base cases and Lemma 1 implies that $Post^{0+}(I) \subseteq S$. The problem is safe. □

Theorem 2. *If* `IC3Sync()` *returns* $\top$*, then the problem is safe. If* `IC3Sync()` *returns* $\bot$*, then the problem is unsafe.*

Proof. If `IC3Sync()` returns $\bot$, then some call to $\texttt{stregthen(}\mathbf{F}_i, \mathbf{K}\texttt{)}$ returns with $bug = \top$. As in the case of IC3, this implies that there exists a sequence $\langle (m_0, 0), (m_1, 1), \ldots, (m_{\mathbf{K}-2}, \mathbf{K}-2)\rangle$ such that:

$$m_0 \models I \wedge m_{\mathbf{K}-2} \models \neg S \wedge \forall i \in [0, \mathbf{K}-2) \centerdot m_i \wedge T \wedge m'_{i+1} \neq \bot \tag{2}$$

This sequence leads to a counterexample. The problem is unsafe. If `IC3Sync()` returns $\top$, then from lines 85–86, we have $\exists j \in [1, \mathbf{K}-2] \centerdot \forall i \in [1, n] \centerdot \mathbf{F}_i[j] = \emptyset \implies \exists j \in [1, \mathbf{K}-2] \centerdot f(j) = f(j+1)$. This, together with $\mathcal{I}_6$, the check for base cases and Lemma 1 implies that $Post^{0+}(I) \subseteq S$. The problem is safe. $\square$

Theorem 3. *If* `IC3Async()` *returns* $\top$*, then the problem is safe. If* `IC3Async()` *returns* $\bot$*, then the problem is unsafe.*

Proof. If `IC3Async()` returns $\bot$, then some call to `stregthen()` returns with $bug = \top$. As in the case of IC3, this implies that there exists a sequence $\langle (m_0, 0), (m_1, 1), \ldots, (m_{\mathbf{K}-2}, \mathbf{K}-2)\rangle$ such that:

$$m_0 \models I \wedge m_{\mathbf{K}-2} \models \neg S \wedge \forall i \in [0, \mathbf{K}-2) \centerdot m_i \wedge T \wedge m'_{i+1} \neq \bot \tag{3}$$

This sequence leads to a counterexample. The problem is unsafe. If `IC3Async()` returns $\top$, then from lines 112–113, we have $\exists j \in [1, \mathbf{K}-2] \centerdot \forall i \in [1, n] \centerdot \mathbf{F}_i[j] = \emptyset \implies \exists j \in [1, \mathbf{K}-2] \centerdot f(j) = f(j+1)$. This, together with $\mathcal{I}_9$, the check for base cases and Lemma 1 implies that $Post^{0+}(I) \subseteq S$. The problem is safe. $\square$

Theorem 4. *If* `IC3Proof()` *returns* $\top$*, then the problem is safe. If* `IC3Proof()` *returns* $\bot$*, then the problem is unsafe.*

Proof. If `IC3Proof()` returns $\bot$, then some call to `stregthen()` returns with $bug = \top$. As in the case of IC3, this implies that there exists a sequence $\langle (m_0, 0), (m_1, 1), \ldots, (m_{\mathbf{K}-2}, \mathbf{K}-2)\rangle$ such that:

$$m_0 \models I \wedge m_{\mathbf{K}-2} \models \neg S \wedge \forall i \in [0, \mathbf{K}-2) \centerdot m_i \wedge T \wedge m'_{i+1} \neq \bot \tag{4}$$

This sequence leads to a counterexample. The problem is unsafe. If `IC3Async()` returns $\top$, then some call to `propProof()` returns with $safe = \top$. Then, from the check at lines 163, and the fact that $C_1 \wedge C_4$ holds at line 163, we know that Π is an inductive invariant that implies S. The problem is safe. $\square$

Theorem 5. *All three parallel variants of* IC3 *terminate on all inputs.*

Proof. Recall the function f from Figs. 2–4. For any index i, let $|f(i)|$ denote the number of satisfying solutions of $f(i)$. Let us write $\mathbf{K}^*$ to mean $\mathbf{K}$ in the case of IC3SYNC, and $\max(\mathbf{K}_1, \ldots, \mathbf{K}_n)$ in the case of IC3ASYNC and IC3PROOF. It can be shown that the following is an invariant of all IC3PARS.

$$|f(0)| = |I| \wedge \forall j \in [1, \mathbf{K}^* - 1] \centerdot f(j-1) \leq f(j)$$

In other words, $f(0)$ has exactly the same number of solutions as the initial states, and the number of solutions of $f(j)$ grows monotonically with j. Suppose an execution of IC3PAR does not terminate. Then we must reach a point where $\mathbf{K}^* > 2^{|V|}$ and $\forall j \in [1, \mathbf{K}^* - 1] \centerdot \exists i \in [1, n] \centerdot \mathbf{F}_i[j] \neq \emptyset$. But this means that $\forall j \in [1, \mathbf{K}^* - 1] \centerdot f(j-1) < f(j)$. Since $|I| > 0$ (otherwise the algorithm terminates with the check for base cases), we have $|f(\mathbf{K}^* - 1)| > 2^{|V|}$. This is absurd since there cannot be more than $2^{|V|}$ solutions to any formula over V. $\square$

B Statistical Analysis of IC3PAR Portfolios

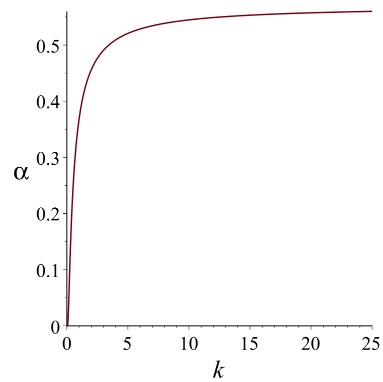


Fig. 11. Plot of $\alpha(k)$ against k.

Lemma 2. *Let* $\alpha(k) = e^{-(\Gamma(1+\frac{1}{k}))^k}$. *Then* $\lim_{k\to\infty} \alpha(k) = e^{-\frac{1}{e^{\gamma}}}$.

Proof. It suffices to show that:

$$\lim_{k\to\infty} (\Gamma(1 + \frac{1}{k}))^k = e^{-\gamma}$$

or, equivalently:

$$\lim_{k\to\infty} k \cdot \ln(\Gamma(1 + \frac{1}{k})) = -\gamma$$

Using the result [2]:

$$\ln(\Gamma(1+z)) = -\gamma \cdot z + \sum_{n=2}^{\infty} \frac{\zeta(n)}{n} \cdot (-z)^n, \qquad \text{if } |z| < 1$$

[2] This result is mentioned at `https://en.wikipedia.org/wiki/Gamma_function`. It can be derived from another result (equation 20 on page 621) in the following paper – Wrench, J. W. Jr. "Concerning Two Series for the Gamma Function." Math. Comput. 22, 617-626, 1968. The paper is available at `http://www.ams.org/journals/mcom/1968-22-103/S0025-5718-1968-0237078-4/S0025-5718-1968-0237078-4.pdf`.

where $\zeta(m)$ is the Riemann zeta function, we get:

$$\begin{aligned}\lim_{k\to\infty} k \cdot \ln(\mathit{\Gamma}(1+\frac{1}{k})) &= \lim_{k\to\infty} k \cdot (-\gamma \cdot \frac{1}{k} + \sum_{n=2}^{\infty} \frac{\zeta(n)}{n} \cdot (-\frac{1}{k})^n) \\ &= -\gamma + \lim_{k\to\infty} k \sum_{n=2}^{\infty} \frac{\zeta(n)}{n} \cdot (-\frac{1}{k})^n \\ &= -\gamma + \lim_{k\to\infty} \sum_{n=2}^{\infty} \frac{\zeta(n)}{n} \cdot (-1)^n \cdot \frac{1}{k^{n-1}}\end{aligned}$$

Since $\lim\limits_{k\to\infty} \frac{1}{k^{n-1}} = 0$ for $n \geq 2$, we immediately get our result:

$$\lim_{k\to\infty} k \cdot \ln(\mathit{\Gamma}(1+\frac{1}{k})) = -\gamma$$

□